ELPI: fast, Embeddable, λProlog Interpreter

Cvetan Dunchev,[1] Ferruccio Guidi,[1] Claudio Sacerdoti Coen,[1] Enrico Tassi[2]

[1] Department of Computer Science, University of Bologna, `name.surname@unibo.it`
[2] Inria Sophia-Antipolis, `name.surname@inria.fr`

Abstract. We present a new interpreter for λProlog that runs consistently faster than the byte code compiled by Teyjus, that is believed to be the best available implementation for λProlog. The key insight is the identification of a fragment of the language, which we call Reduction-Free Fragment, that occurs quite naturally in λProlog programs and that admits constant time reduction and unification rules.

1 Introduction

λProlog is a logic programming language based on an intuitionistic fragment of Church's Simple Theory of Types. An extensive introduction to the language with examples can be found in [8]. Teyjus [9, 10] is a compiler for λProlog that is considered to be the fastest implementation of the language. The main difference with respect to Prolog is that λProlog manipulates λ-tree expressions, i.e. syntax containing binders. Therefore the natural application of λProlog is meta-programming (see [11] for an interesting discussion), including: automatic generation of programs from specifications; animation of operational semantics; program transformations and implementation of type checking algorithms.

Via the Curry-Howard isomorphism, a type-checker is a proof-checker, the main component of an interactive theorem prover (ITP). Indeed the motivation of our interest in λProlog is that we are looking for the best language to implement the so called *elaborator* component of an ITP. The elaborator is used to type check the terms input by the user. Such data, for conciseness reasons, is typically incomplete and the ITP is expected to infer what is missing. The possibility to extend the built in elaborator with user provided "logic programs" (in the form of type classes or unification hints) to infer the missing pieces of information turned out to be a key ingredient in successful formalizations like [3]. Embedding a λProlog interpreter in an ITP would enable the elaborator and its extensions to be expressed in the same, high level, language. A crucial requisite for this plan to be realistic is the efficiency of the λProlog interpreter.

In this paper we introduce ELPI, a fast λProlog interpreter that being written entirely in OCaml can be easily embedded in OCaml softwares, like Coq. In particular we focus on the insight that makes ELPI fast when dealing with binders by identifying a reduction free fragment (RFF) of λProlog that, if implemented correctly, admits constant-time unification and reduction operations. We analyze the role of β-reduction in Section 2 and higher order unification in Section 3; we discuss bound names representations in Section 4; we define the RFF in Section 5 and we asses the results in Section 6.

2 The two roles of β-reduction in λProlog

We introduce λProlog and discuss the role of β-reduction on the Example 1. The λ-term $(\lambda x.xx)$ is encoded as `(lam (x\ app x x))` where `x\F` is the λ-abstraction of λProlog, that binds `x` in `F`, and `lam` is the constructor for object-level abstraction, that builds a term of type $\mathcal{T}$ from a function of type $\mathcal{T} \to \mathcal{T}$, with $\mathcal{T}$ the type of representations of λ-terms. `app` takes two terms of type $\mathcal{T}$ and builds their object-level application of type $\mathcal{T}$. Following the tradition of Prolog, capitals letters denote unification variables.

The second clause for the `of` predicate shows a recurrent pattern in λProlog: in order to analyze an higher order term, one needs to recurse under a binder. This is achieved exploiting the forall quantifier `pi x\G` together with logical implication `F => G`. Operationally: the forall quantifier declares a new local constant `x`, meant to be fresh in the entire program; logical implication temporarily augments the program with the new formula `F` about `x`. Denotationally, these are just the standard rules for introduction of implication and the universal quantifier.

Note that the functional (sub-)term `F` is applied to the fresh constant `x`. Being `F` a function, the β-redex `(F x)`, once reduced, denotes the body of our object-level function where the bound variable is replaced by the fresh constant `x`. The implication is used to assume `A` to be the type of `x`, in order to prove that the body of the abstraction has type `B` and therefore the whole abstraction has type `(arr A B)` (i.e. $A \to B$). Note that, unlike in the standard presentation of the typing rules, we do not need to manipulate an explicit context $\varGamma$ to type the free variables. Instead the assumptions of the form `(of x A)` are just added to the program's clauses, and λProlog takes care of dropping them when `x` goes out of scope. Example: if the initial goal is `(of (lam (w\ app w w)) T)` by applying the second clause we assign `(arr A B)` to `T` and generate a new goal `(of (app c c) B)` (where `c` is the fresh constant substituted for `w`) to be solved with the extra clause `(of c A)` at disposal.

In this first example, the meta-level β-reduction is only employed to inspect a term under a binder by replacing the bound name with a fresh constant. The second example shows a radically different pattern: in order to implement object-level substitution — and thus object-level β-reduction — we use the meta-level β-reduction. E.g. if `F` is `(w\ app w w)` then `(F N)` reduces to `(app N N)`. Note that in this case β-reduction is fully general, because it replaces a name with a term. This distinction is crucial in the definition of the RFF in Section 5.

```
of (app M N) B :-
  of M (arr A B), of N A.
of (lam F) (arr A B) :-
  pi x\ of x A => of (F x) B.
```

```
cbn (lam F) (lam F).
cbn (app (lam F) N) M :- cbn (F N) M.
cbn (app M N) R :-
  cbn M (lam F), cbn (app (lam F) N) R.
```

Example 1: Type checker and Weak CBN for simply typed λ-calculus.

3 Higher Order unification

Unification in the presence of binders raises two problems. First, the absence of most general unifiers (MGUs) makes one of the primitive operations of λProlog very delicate. Second, one has to find a way to avoid captures, i.e. check that unification variables are instantiated with terms containing only bound variable in their scope.

To cope with the absence of MGUs, Dale Miller identified in [7] a well-behaved fragment ($\mathcal{L}_\lambda$) of higher-order (HO) unification that admits MGUs and is stable under λProlog resolution. The restriction defining $\mathcal{L}_\lambda$ is that unification variables can only be applied to (distinct) variables (i.e. not arbitrary terms) that are not already in the scope of the variable. Such fragment can effectively serve as a primitive for a programming language and indeed Teyjus 2.0 is built around this fragment: no attempt to enumerate all possible unifiers is performed, and unification problems falling outside $\mathcal{L}_\lambda$ are just delayed. Many interesting λProlog programs can be rewritten to fall in the fragment. For example, we can make `cbn` of Example 1 stay in $\mathcal{L}_\lambda$ by replacing line 6 (that contains the offending `(F N)` term) with the following code:

```
cbn (app (lam F) N) M :- subst F N B, cbn B M.
subst F N B :- pi x\ copy x N => copy (F x) B.
copy (lam F1) (lam F2) :- pi x\ copy x x => copy (F1 x) (F2 x).
copy (app M1 N1) (app M2 N2) :- copy M1 M2, copy N1 N2.
```

The idea of `subst` is that the term `F` is recursively copied in the following way: each bound variable is copied in itself but for the top one that is replaced by `N`. The interested reader can find an longer discussion about `copy` in [8, page 199]. The `of` program falls naturally in $\mathcal{L}_\lambda$, since `F` is only applied the fresh variable `x` (all unification variables in a λProlog program are implicitly existentially bound in front of the clause, so `F` does not see `x`). The same holds for `copy`.

To correctly implement HO unification, even in the restricted $\mathcal{L}_\lambda$ fragment, one typically tracks the *level* of unification variables an fresh constants. The proof theoretic interpretation of a λProlog execution as an intuitionistic proof gives the following reading: unification is taking place under a mixed prefix of $\forall$ and $\exists$ quantifiers; their order determines if a unification variable (an existential) can be assigned to (proved by) a term that contain a universally quantified variable. E.g. $\forall x, \exists Y, Y = x$ is always provable while $\exists Y, \forall x, Y = x$ is not. Whenever a clause is used, its unification variables are declared at a level that corresponding the length of the current "context", and whenever a fresh constant is created (for the `pi` quantifier) the context is extended and the constant is placed at such level. From now on we will write levels in superscript. If we run the program `(of (lam f\lam w\app f w) T)` after two steps the goal is `(of (app c`^{1} `d`^{2}`) T`^{0}`)`. Replacing `f` and `w` by fresh constant annotated with a level is the implementation technique adopted by Teyjus to make unification able to check levels correctly. As an optimization Teyjus performs the beta reduction in a lazy way using an explicit substitution calculus. In the RFF we will be able to completely avoid such substitution.

rephrase with symbols

sketched

4 Bound variables

The last missing ingredient to define the RFF and explain why it can be implemented efficiently is to see how systems manipulating λ-terms accommodate α-equivalence. Bound variables are not represented by using real names, but canonical "names" (or better numbers). De Bruijn introduced two, dual, naming schemas for lambda terms in [2]: indexes (DBI) and levels (DBL). In the former a variable is named n if its binder is found by crossing n binders going in the direction of the root. In the latter a variable named n is bound by the n-th binder one encounters in the path from the root to the variable. In the following table we write the term $\lambda x.(\lambda y.\lambda z.f\ x\ y\ z)\ x$ and its reduct in the two notations:

$$\begin{array}{rl} \text{Indexes:} & \lambda x.(\lambda y.\lambda z.f\ x_2\ y_1\ z_0)\ x_0 \to_\beta \lambda x.\lambda z.f\ x_1\ x_1\ z_0 \\ \text{Levels:} & \lambda x.(\lambda y.\lambda z.f\ x_0\ y_1\ z_2)\ x_0 \to_\beta \lambda x.\lambda z.f\ x_0\ x_0\ z_1 \end{array}$$

In both notations when a binder is removed and the corresponding variable substituted some "renaming" (called lifting) has to be performed. The DBI convention is way more popular than the DBL one and Teyjus indeed adopts that schema. We believe the popularity of DBI comes from the fact that weak head normalization is easier to code: the argument of the redex, being at the top level of the term, is always closed and hence invariant by lifting.

In ELPI we chose DBL because of the following two properties:

DBL1 the name of an occurrence of a variable does not depend on the (extra) context under which it occurs, i.e. x is always named x_0;

DBL2 when β-reduction occurs under a context the variables bound in such context do not change name. In our example, since the reduction occurs under the binder for x, x is named x_0 in the initial and in the reduct term.

Another way to put it is that variables already pushed in the context are treated *exactly as constants*, and their name is the level at which the occur in the context. From now on we subscript variables with their name in DBL convention.

5 The Reduction-Free Fragment.

λProlog is a truly higher order language: even clauses can be passed around, unified, etc. Nevertheless this plays no role here, so we exclude from the syntax of terms the one of formulas.

$$t ::= x_i \mid X^j \mid \lambda x_i.t \mid t\ t$$

Since variables follow the DBL representation, we don't have a case for constants: when $i < 0$ then x_i represents a global constant, like `app` or `lam` in Example 1. Since the level of a variable completely identifies it, when we write $x_i \dots x_{i+k}$ we mean k distinct bound (i.e. $i \geq 0$) variables. The superscript j annotates unification variables with their visibility range ($0 \leq j$, since all global constants are in range). A variable X^j has visibility of all names strictly smaller than j. E.g. X^1 has visibility only of x^0, and X^3 has visibility of $\{x^0, x^1, x^2\}$. Technically a binder needs no name when bound variables are named following a De Bruijn convention. Still we write it to ease reading.

Definition 1 (RFF) *A term is in the Reduction Free Fragment iff every occurrence of a unification variable X^j is applied to $x_j \dots x_{j+k-1}$ for $k >= 0$.*

Note that when k is 0 the variable is not applied. Another way to put it is that a term is in the RFF iff all unification variables see a (complete, no hole) prefix of the λProlog context seen as a ordered list. Examples: $X^2\ x_2\ x_3$ and X^2 are in the fragment; $X^2\ x_3$ and $X^2\ x_3\ x_2$ are not.

Observe that the programs in Example 1 (when `cbn` is rewritten to be in the pattern fragment as in Section 3) are in the RFF. Also, every Prolog program is in the RFF. As we will see in Section 6, a type-checker for a dependently typed language and evaluator based on a reduction machine are also naturally in RFF, showing that, in practice, the fragment is quite expressive.

Property 1 (Decidability of HO unification) *Being the RFF included in the pattern-fragment, higher order unification is decidable for the RFF.*

The most interesting property of the RFF, which also justify its name, is the following one.

Property 2 (Constant time head β-reduction) *Let σ be a* valid *substitution for existentially quantified variables. Then the head normal form of $(X^j\ x_j \dots x_{j+k-1})\sigma$ can be computed in constant time.*

A valid substitution assigns to X^j a term t of the right type (as in simply type lambda calculus) and that has all free variables visible by X^j (all x_i are such that $i < j$). Let $X^j\sigma = \lambda x_j.\dots\lambda x_{j+n}.t$. Then

$$(X^j\ x_j \dots x_{j+k-1})\sigma = \begin{cases} t\ x_{j+n+1} \dots x_{j+k-1} & \text{if } n+1 < k \\ \lambda x_{j+k}.\dots\lambda x_{j+n}.t & \text{otherwise} \end{cases} \qquad (1)$$

Thanks to property **DBL2**, Equation 1 is *syntactical*: no lifting of t is required. Hence the β-reductions triggered by the substitution of X^j take constant time.

Property 3 (Constant time unification) *A unification problem of the form $X^j\ x_j \dots x_{j+k-1} \equiv t$ can be solved in constant time.*

The unification problem $X^j\ x_j \dots x_{j+k-1} \equiv t$ can always be rewritten as two simpler problems: $X^j \equiv \lambda x_j.\dots\lambda x_{j+k-1}.Y^{j+k}$ and $Y^{j+k} \equiv t$ for a fresh Y. The former is a trivial assignment that requires no check. The latter can be implemented in constant time if: no occur-check is needed for X and if one caches in the terms the level of the highest free variable. Avoiding useless occur-check is a typical optimization of the Warren Abstract Machine (WAM), e.g. when X occurs linearly in the head of a clause. Caching in the terms the maximum free level is economical in terms of space (just one integer) and is something one typically pre-computes on the input term in linear time. With such maximum level l at hand the problem $Y^{j+k} \equiv t$ can be decided by simply comparing $j + k$ with l. These properties enable us to implement the operational semantics of `pi` in constant time for terms in the RFF.

We detail an example. The first column gathers the fresh constants and extra clauses. The second one shows the current goal(s) and the program clause that is used to back chain.

Context	Goals and refreshed program clause
	of (lam x_0\lam x_1\app x_0 x_1) T^0
	of (lam F^0) (arr A^0 B^0) :- pi x_0\ of x_0 A^0 => of (F^0 x_0) B^0
x_0;(of x_0 A^0)	of (lam x_1\app x_0 x_1) B^0
	of (lam G^1) (arr C^1 D^1) :- pi x_1\ of x_1 C^1 => of (G^1 x_1) D^1
x_0;(of x_0 A^0)	of (app x_0 x_1) D^0
x_1;(of x_1 C^0)	of (app M^2 N^2) S^2 :- of M^2 (arr R^2 S^2), of N^2 R^2
x_0;(of x_0 A^0)	of x_0 (arr R^2 S^0), of x_1 R^2
x_1;(of x_1 C^0)	of x_0 A^0 , of x_1 C^0

After the first step we obtain F^0:= x_0\lam x_1\app x_0 x_1; T^0:= arr A^0 B^0; the extra clause about x_0 in the context and a new subgoal. Note that the redex (F^0 x_0) is in the RFF and thanks to Equation 1 head normalizes in constant time to (lam x_1\app x_0 x_1). The same phenomenon arises in the second step, where we obtain G^1:= x_1\app x_0 x_1 and we generate the redex (G^1 x_1). Unification variables are refreshed in the context under with the clause is used, e.g. C is placed at level 1 initially, but in consequence to a unification step they may be *pruned* when occurring in a term assigned to a lower level unification variable. Example: unifying B^0 with (arr C^1 D^1) prunes C and D to level 0.

The choice of using DBL for bound variables is both an advantage and a complication here. Clauses containing no bound variables, like (of x_0 A^0), require no processing thanks to **DBL1**: they can be indexed as they are, since the name x_0 is stable. The drawback is that clauses with bound variables, like the one used in the first two back chains, need to be lifted: the first time the bound variable is named x_0, while the second time x_1. Luckily, this renaming, thanks property **DBL1** can be performed in constant time using the very same machinery one uses to refresh the unification variables. E.g. when the WAM unifies the head of a clauses it assigns fresh stack cells: the clause is not really refreshed and the stack pointer is simply incremented. One can represent the locally bound variable as an extra unification variable, and initialize, when pi is crossed, the corresponding stack cell to the first x_i free in the context.

resurrect Claudio's example?

Stability of the RFF. Unlike $\mathcal{L}_\lambda$, the RFF is not stable under λProlog resolution: a clause that contains only terms in the RFF may generate terms outside the fragment because of the lifting phenomenon explained the previous paragraph. Therefore an implementation must handle both terms in the RFF, with their efficient computation rules, and terms outside the fragment. Our limited experience so far, however, is that several programs initially written in the fragment remains in the fragment during computation, or they can be slightly modified to achieve that property.

6 Assessment and conclusions

We asses the performances of ELPI on a set of synthetic benchmarks and a real application. Synthetic benchmarks are divided into three groups: first order programs from the Aquarius test suite (the crypto-multiplication, μ-puzzle, generalized eight queens problem and the Einstein's zebra puzzle); higher order programs falling in the RFF fragment; and an higher order program falling outside RFF taken from the test suite of Teyjus normalizing expressions in the SKI calculus.

The programs in the RFF are respectively type checking lambda terms using the `of` program of Example 1 and reducing expressions like 5^5 using Church numerals using a call by value strategy. `lambda3` was specifically conceived to measure the cost of moving under binders.

Test	ELPI		Teyjus		ELPI/Teyjus	
	time (s)	space (Kb)	time (s)	space (Kb)	time	space
crypto-mult	3.48	27,632	6.59	18,048	0.52	1.53
μ-puzzle	1.82	5,684	3.62	50,076	0.50	0.11
queens	1.41	108,324	2.02	69,968	0.69	1.54
zebra	0.85	7,008	1.89	8,412	0.44	0.83
typeof	0.27	8,872	5.64	239,892	0.04	0.03
reduce_cbv	0.15	7,248	11.11	57,404	0.01	0.12
reduce_cbn	0.33	8,968	0.81	102,896	0.40	0.08
reduce_cbv_nocopy	...	...	11.25	50,892	...	...
reduce_cbn_nocopy	...	...	0.52	70,760	...	...
SKI	1.32	15,472	2.68	8,896	0.49	2.73

The table shows that ELPI shines on programs in the RFF, and compares well outside it. It is hard to understand why ELPI is faster than Teyjus outside RFF, since all the optimizations we used are inspired by the literature about the WAM, on which Teyjus is also based. Our guess is that the OCaml garbage collector, well known for its efficiency, is responsible for the difference. Especially because we heavily optimized our code to lower the pressure on it to let it perform at the best of its capabilities.

6.1 A Relevant Test Case: the "Grundlagen" Verified

As a relevant test case for ELPI, we implemented in the Reduction-Free Fragment a validator for the latest version of the formal system $\lambda\delta$, and used this validator to verify the "Grundlagen" [13] translated in a Pure Type System [1].

The formal system $\lambda\delta$ [4], improved in [5, 6], is a framework that embeds some former typed λ-calculi including $Aut-QE$, the Automath dialect in which the "Grundlagen" was originally written, and Pure Type Systems like λC.

Current verification algorithms for typed systems follow a well-established pattern prescribing a reduction machine to compute weak head normal forms, a comparator to assert convertibility by levels, and a checker responsible for type inference. The verification algorithm for $\lambda\delta$ deviates slightly from this pattern in that type checking is replaced by validation, and in that type inference is delegated to the (extended) reduction machine. The performance benefits of this approach are documented in [6]. We wish to recall that type checking a term means asserting that this term has a specified type, whereas validating a term means asserting that this term has some unspecified type.

A validator for $\lambda\delta$, named Helena, has been implemented in Caml, and our λProlog implementation follows it closely. Nevertheless, the λProlog code is much simpler that the corresponding Caml code (441 lines), and consists of just 52 clauses. In particular, the built-in environment of the λProlog engine is used in place of the environments explicitly implemented in Caml.

The translated "Grundlagen" is a theory comprising 32 declarations and 6879 definitions, for a total of 6911 items. Each item is a term to be verified, written in the raw syntax of λC with constants type casts. The sort $\Box$ never appears explicitly and the sorts *set* or *prop* are used in place of the sort $*$.

The term in each definition is a type cast in a context of λ-abstractions corresponding to the "block openers" of $Aut-QE$. On the other hand, the term in each declaration (that is, a type) is given in a context of Π-abstractions.

Overall, the tree representation of these terms consists of 754579 nodes. This huge amount of data seems to overwhelm the capabilities of Teyjus, which is limited by design restrictions. Therefore, as of now, we can verify with this system just the first 255 items of the "Grundlagen".

On the other hand, we can present the full "Grundlagen" to Coq [6].

In the tables below, we compare preprocessing (Pre), and verification (Ver) for Helena, ELPI, Teyjus, and Coq. Depending on the system, verification refers to validation, type checking, and (bytecode) execution, while preprocessing accounts for parsing, internalization, compilation, linking, and refinement.

User time range (s) for 31 runs (255 items only)		
Task	ELPI	Teyjus
Pre	00.07 to 00.09	03.24 to 03.31
Ver	00.20 to 00.22	02.13 to 02.26

User time range (s) for 31 runs (all items)			
Task	Helena	ELPI	Coq
Pre + Ver, compiled	01.14 to 01.18	not applicable	24.26 to 24.43
Pre + Ver, interpreted	08.73 to 08.77	27.52 to 27.82	94.18 to 95.78
Ver, interpreted	05.54 to 05.58	21.15 to 21.45	51.63 to 53.76

References

1. H.P. Barendregt. Lambda Calculi with Types. *Osborne Handbooks of Logic in Computer Science*, 2:117–309, 1993.
2. N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Selected Papers on Automath [12]*, pages 375–388. North-Holland, 1994.

3. G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A Machine-Checked Proof of the Odd Order Theorem. In *Proceedings of ITP 2013*, volume 7998 of *LNCS*, pages 163–179, July 2013.
4. F. Guidi. The Formal System $\lambda\delta$. *Transactions on Computational Logic*, 11(1):5:1–5:37, online appendix 1–11, November 2009.
5. F. Guidi. The Formal System $\lambda\delta$ Revised, Stage A: Extending the Applicability Condition. CoRR identifier 1411.0154, November 2014. Submitted to Transactions on Computational Logic, ACM (available at <http://lambdadelta.info/>).
6. F. Guidi. A Verified Translation of Landau's Grundlagen" from Automath into a Pure Type System, via $\lambda\delta$, February 2015. Submitted to Journal of Formalized Reasoning, University of Bologna (available at <http://lambdadelta.info/>).
7. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1:253–281, 1991.
8. D. Miller and G. Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012.
9. G. Nadathur and D. J. Mitchell. System description: Teyjus - A compiler and abstract machine based implementation of lambda-prolog. In *Proceedings of CADE-16*, pages 287–291, 1999.
10. X. Qi. An implementation of the language lambda prolog organized around higher-order pattern unification. *CoRR*, abs/0911.5203, 2009.
11. O. Ridoux. *Lambda-Prolog de A a Z... ou presque*. Habilitation à diriger des recherches, Université de Rennes 1, 1998.
12. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1994.
13. L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the automath system*, volume 83 of *Mathematical Centre Tracts*. 1979.